# CSE 498N/598N Home work project: P2P Overlay Networks for Data Distribution

For the first three home work projects, we will implement a peer to peer scheme for distributing large data objects (similar to BitTorrent [http://bitconjurer.org/BitTorrent/]). The system is designed to distribute large objects (like the latest Linux kernel) to a large number of peers. We assume that many peers have copies of the object; the goal is to find them and request the objects in a distributed fashion from as many peers as necessary. Many of these peers are transient. The functionality that we will implement is the ability to search and resiliently serve the values (given a certain key, find and download the value in the face of node failures). Each peer listens on a UDP port and maintains simple *key:value* tuples. *Key* is the name of the file and *value* is the actual data. There are a number of components necessary to implement this project. First we need to name each peer that are participating in our data distribution, build an overlay where nodes know one another. Hopefully we can reach any peer on the system by asking other peers in turn. We can use this overlay to search for the nodes that actually contain the data that we are interested in. Given a number of such peers, we would like to receive data in a distributed fashion such that we can recover from node failures as well as leverage the overall network bandwidth available on the overlay. Once you have implemented the system, you can port your system to the planet-lab infrastructure (www.planetlab.org) to implement your system across the wide area Internet. For testing purposes (and for HW 1 through 3), you can use the FreeBSD 5.2 cluster in Cushing Rm 208 (gateway13.cse.nd.edu to gateway18.cse.nd.edu) to implement your system.

## 1 Home work project #1 - Locating and creating an overlay

### 1.1 Peers

The first problem that we need to solve is to name and identify the peers. Each peer independently chooses its name. For this project, we will use *node IP:port* as the name of each peer. Each peer will listen on this port and provide service to other peers (answering file queries, service the files etc.) on this port; you choose this specific port. You could run a number of peers on the same machine by choosing different port numbers for each one of them.

The peers will maintain *key:value* tuples. The peers will provide the following interface for services (note that the services are described in a 'C' like pseudo function call. You are free to implement it in a fashion that is convenient for you):

- **get(key)** This service will return the *value* associated with a given *key*. The key should be among the keys listed in the *list* service.
- **list()** This service will list all the keys that are available at the peer (set using earlier *set* operations).

### 1.2 Location service

Next, these peers will maintain information about other peers that are currently online. In general, there is a limit to the number of peers that you can maintain information about. For this project, the peers will be restricted to manage location information about **two** other peers. For debugging purposes, the peers will continue to print location information such as peers entering and leaving the system.

Peers identify each other by exchanging the *identification_t* structure. For this project, you will exchange the name, the Internet port number and Internet address where you can be reached in the *identification_t* structure. Peers can also maintain the round trip times to the different peers in order to choose the "best" peers. You can refer to the COMPUTER NETWORKS book by *W. Richard Stevens* [2] for sample code on using network system calls. The sample code from this book is available online at http://www.kohala.com/start/unpv12e.html.

```
typedef struct identification {
  char name[32];               /* Name of the current client */

  // Specify how we can be contacted.
  in_addr_t  location;       /* IP address of the client */
  in_port_t  port;           /* port where the client is listening */
} identification_t;
```

There are a number of different ways to locate other peers. For this project, you will use peer-to-peer techniques to locate other peers (and not a centralized approach). The peers will directly locate other peers without any centralized data structures. Peers can utilize multicasting (all peers listen on different multicast channels) or broadcasting to identify other peers. Peers can broadcast a query asking other peers to identify themselves or new peers can initially broadcast their identity in order to join the community. You should be able to locate instances of your own peer running on different hosts (you would have to explicitly start a number of peers). You may also be able to locate peers developed by your classmates.

### 1.3 Sample output:

This is a sample run for how you might print other peers entering and leaving the system.

1/9/2004 10:30 'John Doe' ENTER darwin.cc.nd.edu:6780 20 msec

1/9/2004 10:35 'John Doe' MAINTAIN darwin.cc.nd.edu:6780 30 msec

1/9/2004 10:36 'John Doe' LEAVE darwin.cc.nd.edu.edu:6780

1/9/2004 10:40 'Jane Doe' ENTER wizard.cse.nd.edu:6003 100 msec

## 2 HWP #2: Service the tuples: Query/response routing

The next step is to be able to access *key:values* that are available in peers that may not be directly known to the current peer. When contacting remote peers, you are only allowed to directly contact a peer that you already know about. All other peers should be queried through peers that you know about. For example, suppose we know about peer FOO on host1:port1 and we get a request for BAR on host1:port1, we have to send the request to FOO which might forward the request to BAR. The peers will provide these services:

- **search(searchKey, hopCount)** If the requested key *searchKey* is available in the peer, the identity of the peer (in a printable *hostIP:port* format) is sent back. If the key *searchKey* was not available, a recursive **searchget** is invoked by this peer (on behalf of the requestor) on all the peers that it knows of (restricted to two for this project). Every such forwarding decrements the hopCount. Once the hopCount reaches 0 without successfully finding the file, the system returns an error message.

  Note that your implementation might return multiple values for the same key. It might also return an KeyNotFound error, even though there is a path available from the source to the destination.

- **rget(peerHost, peerPort, key)** This service will return the *value* associated with a given *key* in the peer at peerHost:peerPort. You are only allowed to directly contact the peer at peerHost:peerPort if your own internal routing tables know about this peer. If you do not have direct knowledge of the peer, you should forward it to a peer that you know. The responses also follow a similar pattern.

# 3 Home work project #3: Reliable communications over unreliable channels

Over the course of the semester, we spent considerable time developing network technologies that provide reliable communication channels (TCP) over unreliable channels (IP). For this home work project, we will build such a reliable communication mechanism over an unreliable UDP based stream (along with dummynet interface to control and tweak the network parameters). We will integrate this reliable communication mechanism into our peers to provide reliable communication among the various peers.

You will develop an reliable network mechanism that transfers large files (e.g. 5 MB). You have to deal with cross-flows (from other UDP and TCP traffic). You will plot the network throughput, the time taken to transfer the file and the amount of data transferred (both the forward direction and acknowledgment traffic). You will use dummynet to simulate interesting network scenarios. A presentation describing dummynet is available at http://info.iet.unipi.it/~luigi/bsdcon01/dummynet/.

The grading will be done as follows: If you implement basic ARQ functionality, then you will receive 60% points. Every additional feature that you implement will get you 10% more points. You could receive more than 100% points for more complex (and correct) implementations. Untested (but potentially correct implementations) will not receive any credit. At the end, we will compare your results with other implementations by your colleagues. The best implementation (as measured by the overall throughput and amount of acknowledgment traffic) will receive an extra credit (of 10% points). Specifically, we will:

1. Develop a program that will reliably (in-order, delivered once semantics) transfer 5 MB of data from one machine to another machine (within the class network cluster). You will use a maximum packet size of 100 bytes. You will use UDP as the underlying network mechanism and implement TCP functionality. You need not implement the TCP connection open (SYN) and connection shutdown (FIN) mechanisms. Your implementation should adaptively measure the RTT. Implement some/all of the following techniques:

   - Slow start
   - Fast recovery
   - SACK
   - DSACK
   - delayed ACK
   - other techniques used in TCP or discussed in class

   Clearly identify the features that you had implemented to receive proper credit.

   The next few features are necessary for the proper testing and simulation setup of your program and as such carry no specific grade:

2. Develop a UDP program that will continuously use half the available bandwidth (for example, for a 1 Mbps link, send 625 - 100 byte packets per second ($625 * 100 * 8 = 500kbps$)). You will run this program simultaneously to provide the cross flows.

3. Use dummynet [1] to simulate the following network conditions between the source and destination (full duplex settings; 8 combinations):

   (a) 100 kbps and 1 Mbps.

   (b) delays of 100 msec and 1 sec.

   (c) packet loss of 1% and 10%.

4. Run two instances of your program along with the UDP program and measure the instantaneous throughput and plot the throughput measured with time (in 100 msec granularity). You developed such a throughput measurement tool for your HWA #1. Also measure the amount of traffic on the wire (both directions, including retransmissions, ACK packets etc).

### 3.1 Integration

Now, integrate your reliable delivery system into your entire system. We will port your system to the planet-lab infrastructure (www.planetlab.org) to implement your system across the wide area Internet. We will measure the end-to-end throughput, system resiliency by running your system across the wide area.

## 4 Submission guidelines

You are free to choose your program implementation language. In general, choose a language that you know and are comfortable with. Submit your project, along with a succinct report called REPORT.txt (plain text is fine) describing your approach, the merits of your approach and compilation instructions. You will turn in your complete project as a single tar file. You can use the course AFS dropbox at
/afs/nd.edu/coursesp.04/cse/cse498n.01/dropbox/$< loginid >$.
Evaluate your implementations on the following issues in the REPORT.txt:

1. **interoperability:** How does your peer recognize your friend in a different host/operating system. For example, if you are working in wizard [Sun Sparc machine running Solaris], can you identify your friend in the dorm using a Pentium III running Linux?
2. **consistency:** How quickly do peers realize when a peer crashes so that the display that you get accurately reflects the peers that are currently online?
3. **scalability:** If your peer suddenly becomes popular because of the files that your provide, how much load can you tolerate before your system crashes? Does your service degrade gracefully? Are you immune to denial-of-service attacks? (wherein, peers repeatedly open connections to you to prevent you from servicing other, legitimate users)
4. **robustness:** How reliable is your system against failures? Does your peer recognize forwarding loops? (wherein the requests are forwarded around in a loop without making progress towards the destination)

## References

[1] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.

[2] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*, volume 1 of *ISBN 0-13-490012-X*. Prentice Hall, 2 edition, 1998. Sample code from this book is available at http://www.kohala.com/start/unpv12e.html.